Leveraging Years of Classical Compiler Expertise for Quantum Computing

Patrick Hopf, *Technical University of Munich, 80333, Munich, Germany, and MQSC, 85748, Garching, Germany*
Yannick Stade and Damian Rovara, *Technical University of Munich, 80333, Munich, Germany*
Daniel Haag, *Technical University of Munich, 80333, Munich, Germany, and MQSC, 85748, Garching, Germany*
Matthias Reumann, *Technical University of Munich, 80333, Munich, Germany*
Lukas Burgholzer and Robert Wille, *Technical University of Munich, 80333, Munich, Germany, and MQSC, 85748, Garching, Germany*

Decades of research have gone into classical compiler design, producing sophisticated frameworks that transform high-level programs into efficient machine code. Quantum computing faces the same challenge: translating quantum algorithms into hardware-executable instructions. A classical-first approach extends established compiler frameworks with quantum concepts, treating quantum processors as specialized accelerators within larger computing systems. This article presents multilevel intermediate representation (MLIR) as an effective foundation for this approach and demonstrates its use in the Munich Quantum Software Stack, which connects high-level applications to diverse quantum hardware through a modular, high-performance computing (HPC)-integrated pipeline. MLIR dialects support multiple abstraction levels while providing built-in parsing, transformation, and optimization infrastructure. Quantum programs are represented through complementary dialects: one explicitly encodes the circuit's dataflow graph, while the other mirrors established languages for seamless tool interoperability. The resulting framework, the Munich Quantum Toolkit Compiler Collection, supports compiler passes for both flat circuits and structured control flow. It is available at https://github.com/munich-quantum-toolkit/core.

Compiler design is one of the great success stories of computer science. Over many decades, researchers and engineers have poured tremendous effort into frameworks that automatically transform high-level source code into efficient machine instructions. Projects such as LLVM[1] underpin virtually every piece of modern software, embodying many years of collective expertise in register allocation, instruction scheduling, and optimization. This maturity did not arise by accident: It is the product of rigorous theoretical foundations and community-wide convergence on shared abstractions—most notably, the *intermediate representation* (IR), a language-agnostic program description that decouples language frontends from hardware back ends and makes optimization passes reusable across both.

Quantum computing, as introduced in the previous article in this series,[2] is expected to be delivered through specialized accelerators coupled with existing scientific-computing environments rather than by standalone quantum systems. As a result, researchers, high-performance computing (HPC) practitioners, and software engineers face a practical challenge: how to integrate quantum hardware into established software stacks and workflows without creating an entirely separate software ecosystem.

© 2026 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/
Digital Object Identifier 10.1109/MCSE.2026.3718606
Date of current version 23 September 2026.

At the heart of this challenge lies the compilation infrastructure that connects high-level programs with hardware-specific execution. For practitioners, the compiler is a crucial piece of this puzzle: Without sophisticated compilation, the full potential of quantum computers cannot be exploited, as high-level algorithms cannot be efficiently translated to hardware-executable instructions. Fortunately, quantum computing faces many of the same fundamental challenges that originally motivated classical compiler development. Both domains must bridge the gap between human-readable algorithms and hardware-executable instructions while dealing with limited resources, instruction scheduling, and mixed levels of abstraction. Yet two fundamentally different strategies have emerged for building the required compilation infrastructure.

The first strategy, which we call *quantum-first*, reconstructs the compilation stack essentially from scratch. Early quantum IRs such as OpenQASM[3,4] established qubits and gates as first-class primitives, and frameworks such as Qiskit[5] and PennyLane[6] followed with their own circuit representations and custom transformations. While this approach succeeded in capturing the flat-circuit programming model, it ran into fundamental limitations when developers began to represent structured control flow, dynamic circuits, or hybrid quantum–classical workflows—classical elements were either absent or added as ad hoc extensions. Critically, when classical control flow is involved in quantum programs, such as in quantum error correction routines with mid-circuit measurements and conditional gates, quantum-first approaches fail to deliver the desired performance.

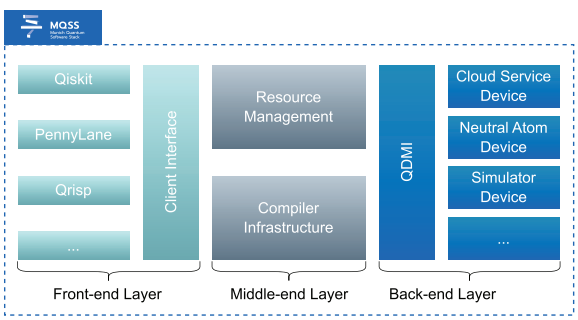

FIGURE 1. High-level overview of the MQSS. Programs can enter the stack through frontend adapters and a shared client interface; they are scheduled and compiled in the middle-end layer and are dispatched via QDMI to quantum backend devices for execution.

The second strategy, *classical-first*, takes the opposite view: Instead of reimplementing decades of compiler infrastructure, it extends proven classical frameworks with quantum concepts, treating the quantum processor as a specialized accelerator within a classical environment. In this article, we consider this approach and demonstrate how the *multi-level intermediate representation* (MLIR)[7] provides an ideal foundation, resulting in the *MQT Compiler Collection* (`mqt-cc`)[8] presented here. We demonstrate how this architecture enables quantum hardware to be integrated into established scientific-computing environments. To illustrate this integration, `mqt-cc` is embedded within the *Munich Quantum Software Stack* (MQSS), a modular, HPC-integrated quantum software ecosystem that connects high-level applications to diverse quantum hardware (cf. Figure 1).

THE QUANTUM SOFTWARE STACK

Before diving into the compilation framework itself, we first sketch the broader software stack in which its operation is showcased in the following. Quantum computers function as accelerators within a larger classical computing environment, much the way GPUs are integrated into HPC clusters today. Bridging a diverse ecosystem of high-level applications and frameworks to a heterogeneous landscape of quantum hardware platforms requires a complete software stack—not just a compiler.

The MQSS[8] is one concrete realization of such a stack and is shown in Figure 1. It is a modular, open-source ecosystem designed to support hybrid quantum–classical workflows from the moment a user submits a problem description to the moment results are returned from hardware. It is structured in three layers:

- The *front-end layer* accepts programs from frameworks such as Qiskit[5] and PennyLane,[6] as well as from domain-specific tools for users with expertise in application areas such as chemistry or finance. Dedicated frontend adapters pass these programs to a shared client interface, which forwards them to the runtime system.
- The *middle-end layer* contains the central runtime component for resource management and compilation. It allocates quantum resources, orchestrates job execution, and invokes the compiler—the central component in this article—to transform input programs into hardware-specific executable representations. Its scheduling and compilation decisions are guided by device

availability, calibration data, and hardware constraints obtained from the back end.

- The *back-end layer* connects the stack to heterogeneous quantum hardware through the quantum device management interface (QDMI).[9] The QDMI provides a standardized interface for session and job management, result retrieval, device–status queries, calibration data, and hardware properties such as gates, error rates, durations, and topology.

At this point, another parallel can be drawn with classical accelerator stacks. A quantum job arriving at the middle-end layer looks, to the scheduler, much like a GPU job, while compilation follows a similarly staged model. Just as CUDA programs may first be compiled to Parallel Thread Execution (PTX) code and only later lowered to device-specific GPU machine code, MQSS first operates on a hardware-independent quantum program representation and performs high-level optimizations. Only after a suitable QPU has been selected does the compiler produce a low-level representation native to that QPU, using device-specific operations, calibration data, and topology constraints. This native representation is submitted through QDMI, which follows the design philosophy of classical hardware abstraction layers by providing a uniform application programming interface regardless of whether the back end is a superconducting processor, a trapped-ion machine, or a neutral-atom device.

Classical software is therefore not a supporting character in quantum application development; it is load-bearing at every level of the stack. This observation motivates the *classical-first* approach of the MQSS and its compiler—the focus of the rest of this article.

MLIR—THE FOUNDATION

Writing a compiler from scratch is a major undertaking: Parsing, type checking, IR design, pass management, code generation, and optimization all require significant engineering investment. The quantum community has built many such tools independently, and the fragmentation shows: Representations are incompatible, passes cannot be shared, and every new framework reinvents the same machinery.

This is precisely the challenge that the MLIR framework,[7] part of the LLVM project, was designed to solve in the classical world; and which is now the foundation on which `mqt-cc`[8] is built. MLIR's central abstraction is the *dialect*: a self-contained vocabulary of operations, types, and attributes tailored to a specific domain or level of abstraction. For software engineers, dialects play a role similar to well-defined interfaces: They separate domain-specific abstractions from their implementation, allowing different components to evolve independently. A machine-learning dialect might define tensor contractions; a hardware dialect might describe register-level instructions; a quantum dialect—as in `mqt-cc`—defines qubits, gates, and measurements. Multiple dialects can coexist within a single program, so quantum operations and classical control flow appear side by side in the same IR.

MLIR ships with a standard library of dialects that cover the most common needs in compiler construction. The `scf` dialect provides `for` loops and `if` conditionals, `arith` handles arithmetic, `func` manages function definitions and calls, `memref` models mutable memory, and `tensor` models immutable arrays. When adding a quantum dialect, a developer builds on top of these and adds only what is genuinely new. Parsing, pretty-printing, type verification, and pass infrastructure come for free.

A *translation* framework handles the boundaries between MLIR and external representations: Programs, e.g., in OpenQASM, are imported into a quantum dialect for optimization, and results can be exported to, e.g., LLVM IR or the quantum intermediate representation (QIR) for execution on simulators or real hardware.

The *multilevel* strategy as part of the framework's name refers to progressive *lowering: A* program starts at a high abstraction level, close to the source algorithm, and is incrementally transformed toward machine code through a series of conversion steps. At each stage, the *conversion framework* replaces operations from one dialect with semantically equivalent

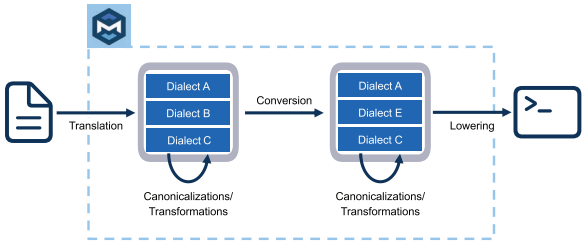

FIGURE 2. Compilation workflow in MLIR. External representations are imported through the translation framework into MLIR dialects, where canonicalization patterns and transformation passes optimize the program. The conversion framework then progressively lowers the program through lower-level dialects before it is exported to an executable representation.

operations from a lower-level dialect, handling type mappings automatically. In parallel, *canonicalization patterns*—local rewrite rules—simplify redundant or unnormalized operations, while *transformation passes* handle more global restructuring. Both mechanisms share the same infrastructure: A pass is declared once in a lightweight TableGen specification, and the framework generates the registration and command-line integration code automatically.

Together, these building blocks (illustrated in Figure 2) constitute a complete compiler framework that the quantum community can extend without reimplementing any of the underlying machinery. That is the core motivation of the classical-first approach; the rest of this article shows what it looks like in practice.

ADDING QUANTUM TO MLIR

With MLIR as the foundation, the question becomes how to best represent quantum programs within it. `mqt-cc` answers this with two complementary dialects, each designed for a different purpose in the compilation pipeline. The quantum circuit optimization (QCO) dialect uses *value semantics*—every gate consumes qubit values and produces new ones—making data dependencies explicit and facilitating optimizations. The quantum circuit (QC) dialect uses *reference semantics*—operations act on named, persistent qubits in place—mirroring the style of languages like OpenQASM and making import and export translations straightforward. Automated conversion passes bridge the two representations, as the following subsections describe.

Representing Quantum Programs

As in software engineering more broadly, choosing an appropriate data representation often eliminates the need for auxiliary data structures and simplifies subsequent algorithms. Due to the employed value semantics, the quantum circuit's representation in QCO is conceptually equivalent to the directed acyclic graph (DAG) that frameworks like Qiskit build internally for circuit analysis. The crucial difference is that in QCO, the DAG is not a separate data structure maintained alongside the program—it *is* the program. The def-use chains of MLIR's single static assignment (SSA) form directly encode the circuit's dependency graph, so all of MLIR's built-in analysis and transformation machinery can be applied to the circuit's dataflow without any translation step.

The value–semantics model also enforces an important physical correctness property. Because quantum information cannot be copied, QCO represents qubits in a linear SSA style: Each qubit value must have exactly one use (cf. Figure 3). Any transformation that would duplicate a qubit value, therefore, produces malformed IR and is rejected by verification. This means that any program passing MLIR's verification is guaranteed to be physically realizable from the linearity standpoint, enabling error detection earlier in the compilation pipeline.

The explicit data flow also has implications for multiqubit operations. A two-qubit gate in QCO consumes values from two separate qubit chains and produces updated values for both, making it immediately visible in the IR whether two gates share a qubit (cf. Figure 3). The practical benefit is that optimization passes become concise and reliable: Detecting whether two operations act on the same qubit in sequence is as simple as checking whether the output of one is the input of the other. This direct correspondence between circuit structure and IR structure is the foundation on which all of `mqt-cc`'s optimizations are built, and it also enables more advanced analyses—computing critical paths, detecting independent subcircuits, reasoning about gate commutativity—all using MLIR's standard infrastructure, with no custom data structures required.

For programs that work with registers of qubits—arrays addressed by index—QCO provides a corresponding abstraction built on MLIR's standard tensor type. Extracting a qubit from a register and inserting an updated qubit back are explicit IR operations, keeping the value semantics invariant intact even for dynamically indexed access patterns.

Connecting Other Tools

In practice, most software systems derive their value not from individual components but from how easily those components interoperate—an idea that is central to MLIR.[10] As powerful as the QCO dialect is for optimization, its value semantics make it cumbersome as an interchange format. Established quantum tools—OpenQASM, Qiskit, Cirq, and most hardware back ends—follow an *imperative* programming model: Qubits are named, persistent resources that operations modify in place, with data flow left implicit in the program order. Translating between this model and QCO's functional representation at every tool boundary would add friction to every integration. This is the role of the QC dialect.

QC uses *reference semantics:* Qubit operations accept qubit references and update the underlying state as a side effect, without consuming or producing new qubit values. A qubit reference persists throughout

the program and is acted upon by a sequence of operations, like in OpenQASM (cf. Figure 4).

The QC dialect serves as both the entry and the exit point of the compilation pipeline. Programs enter, e.g., via OpenQASM and are imported into QC; once optimization in QCO is complete, the result is converted back to QC and can be exported for further optimization with other tools. At the end of the full compilation procedure, the program is usually further lowered to QIR for execution on simulators or physical hardware.

The conversion between QC and QCO is fully automated in both directions and preserves the program's semantics. Converting from QC's reference-based representation to QCO's value-based representation is called *linearization* (Figure 3 ← Figure 4): the conversion pass tracks the latest qubit value associated with each qubit reference and rewires the IR accordingly. The reverse conversion is called *bufferization* (Figure 3 → Figure 4): explicit qubit-value chains are converted back into persistent qubit references updated in program order. This bidirectional design lets `mqt-cc` act as a drop-in optimization engine within existing toolchains, without requiring those toolchains to change.

QUANTUM OPTIMIZATION PASSES

With two complementary dialects in place, we can now see them in action. The strength of the classical-first MLIR approach becomes clearest when implementing optimization passes, where the shared infrastructure dramatically reduces the code a developer must write.

As an example, we consider Hadamard cancelation. The Hadamard gate is self-inverse: $HH = I$. Any two consecutive Hadamard gates acting on the same qubit cancel each other and can be removed without changing the circuit's outcome. While simple, this optimization is a representative example of a broad class of gate-cancelation rules that any practical quantum compiler must support.

In `mqt-cc`, this optimization is implemented as a *canonicalization pattern* in the QCO dialect (cf. Figure 5). Thanks to QCO's value semantics, the pattern logic is direct: We match a Hadamard operation, check whether its output value is the input of another Hadamard, and if so, replace all uses of the second Hadamard's output with the input of the first. Both gates are then dead and removed automatically. The pattern body fits in a dozen lines of C++ since the framework provides the matching and rewriting machinery.

Only little boilerplate is needed around the pattern. A short TableGen declaration specifies the pass name, the dialect it depends on, and a brief description; MLIR then generates the registration and command-line integration code. The pass registers the pattern and invokes `applyPatternsGreedily`, which applies all registered patterns until a fixpoint is reached.

The same infrastructure also supports more general gate-cancelation rules. `mqt-cc` includes an inverse-pair cancelation pass for consecutive gates where one is the inverse of the other, such as T and $T^\dagger$ or S and $S^\dagger$. The pass is implemented once as a generic C++ template, so adding another supported gate pair does not require changing the traversal or pass infrastructure.

Overall, adding a local quantum optimization follows the standard MLIR workflow: Define the pass in TableGen, inherit from the generated C++ base class, implement `runOnOperation`, and register the desired rewrite patterns. The resulting pass integrates with

QCO-Bell.mlir

```
%q0_0 = qco.alloc : !qco.qubit
%q1_0 = qco.alloc : !qco.qubit

%q0_1 = qco.h %q0_0 : !qco.qubit -> !qco.qubit

%q0_2, %q1_1 = qco.ctrl(%q0_1) targets (%arg0 = %q1_0){
   %q0_2 = qco.x %arg0 : !qco.qubit -> !qco.qubit
   qco.yield %q0_2
}:({!qco.qubit},{!qco.qubit})->({!qco.qubit},{!qco.qubit})

%q0_3, %c0 = qco.measure %q0_2 : !qco.qubit
%q1_2, %c1 = qco.measure %q1_1 : !qco.qubit

qco.sink %q0_3 : !qco.qubit
qco.sink %q1_2 : !qco.qubit
```

FIGURE 3. Bell-state program in the QCO dialect using linear SSA value semantics. Each quantum operation consumes qubit values and produces updated qubit values, making the circuit dataflow explicit in the IR.

QC-Bell.mlir

```
%q0 = qc.alloc : !qc.qubit
%q1 = qc.alloc : !qc.qubit

qc.h %q0 : !qc.qubit

qc.ctrl(%q0) targets (%arg0 = %q1) {
   qc.x %arg0 : !qc.qubit
} : !qc.qubit


%c0 = qc.measure %q0 : !qc.qubit -> i1
%c1 = qc.measure %q1 : !qc.qubit -> i1

qc.dealloc %q0 : !qc.qubit
qc.dealloc %q1 : !qc.qubit
```

FIGURE 4. Bell-state program in the QC dialect using reference semantics. Qubit SSA values denote persistent references, and quantum operations update the referenced qubit state in program order.

the command line and the rest of the pipeline without custom driver code.

HYBRID QUANTUM-CLASSICAL OPTIMIZATIONS

As for other hardware accelerators such as GPUs or field-programmable gate arrays, complete quantum applications are hybrid programs in which classical code orchestrates accelerator execution, making modern quantum programs much more than sequences of quantum gates. Quantum error correction uses mid-circuit measurements and feed-forward control, while variational algorithms embed quantum kernels inside classical optimization loops. The control flow itself remains classical: Loop iterations and branch decisions determine when quantum operations are issued by the host or control system, not how the quantum state evolves internally. A compiler for such programs must therefore handle structured control flow directly, rather than only optimizing fully flattened circuits.

In `mqt-cc`, structured control flow comes from MLIR's `scf` dialect, which provides `scf.for` and `scf.while` loops as well as `scf.if` conditionals. These constructs contain nested *regions*, and `QC` and `QCO` operations can appear inside them like ordinary MLIR operations. MLIR traverses regions uniformly, so the same canonicalization patterns apply inside loop bodies, branch bodies, functions, and arbitrary nestings of these constructs. No special quantum control-flow construct is needed.

A common example is preparing an N-qubit GHZ state by applying a Hadamard to the first qubit and then CNOT gates from that qubit to the remaining qubits. This is naturally expressed as an `scf.for` loop over a qubit register. In `QCO`, the register is represented as a tensor of qubit values; each iteration extracts a qubit, operates on it, and inserts the updated value back. This preserves value semantics even for indexed access. If the loop bound is static, MLIR's loop-unrolling pass can produce a flat circuit; if it is dynamic, the program remains well-formed, and optimizations can still apply inside the loop body.

Dynamic reset illustrates the same point for conditionals. A measurement produces a classical result, and an `scf.if` can conditionally apply an X gate to return the qubit to $|0\rangle$. Again, the classical branch structure comes from MLIR, while the quantum operations remain ordinary `QC` or `QCO` operations inside the branch regions.

This shared IR also lets classical and quantum optimizations compose. For example, loop unrolling can expose gate sequences that were separated by iteration boundaries, as shown in Figure 6. A subsequent Hadamard-cancelation pass can then remove the newly adjacent gates. Neither pass needs to know about the other because both operate on the same MLIR module. If unrolling is impossible, e.g., because the loop bounds are dynamic, a dedicated loop-boundary optimization

CancelConsecutiveHadamards.cpp

```
struct CancelConsecutiveHadamardsPattern final : OpRewritePattern<HOp> {
 using OpRewritePattern::OpRewritePattern;
 LogicalResult matchAndRewrite(HOp op, PatternRewriter& rewriter) const override {
  Value out = op.getOutputQubit(0);
  if (!out.hasOneUse()) {
   return failure();
  }
  auto next = dyn_cast<HOp>(*out.user_begin());
  if (!next) {
   return failure();
  }
  rewriter.replaceAllUsesWith(next.getOutputQubit(0), op.getInputQubit(0));
  rewriter.eraseOp(next);
  rewriter.eraseOp(op);
  return success();
 }
};
```

FIGURE 5. Hadamard-cancelation pattern in the `QCO` dialect. The pattern matches a Hadamard whose output qubit value is used by a second Hadamard, then rewires the second output to the first input. The two Hadamard operations become dead and are removed by the rewrite infrastructure.

loop.mlir

```
%c0 = arith.constant 0 : index
%c1 = arith.constant 1 : index
%c2 = arith.constant 2 : index
%q0 = qco.alloc : !qco.qubit

%qf = scf.for %i = %c0 to %c2
      step %c1 iter_args(%q = %q0) -> (!qco.qubit) {
           %q1 = qco.h %q : !qco.qubit -> !qco.qubit
           %q2 = qco.z %q1 : !qco.qubit -> !qco.qubit
           %q3 = qco.h %q2 : !qco.qubit -> !qco.qubit
           scf.yield %q3 : !qco.qubit
}
qco.sink %qf : !qco.qubit
```

after-unroll.mlir

```
%q0 = qco.alloc : !qco.qubit

%q1 = qco.h %q0 : !qco.qubit -> !qco.qubit
%q2 = qco.z %q1 : !qco.qubit -> !qco.qubit
%q3 = qco.h %q2 : !qco.qubit -> !qco.qubit
%q4 = qco.h %q3 : !qco.qubit -> !qco.qubit
%q5 = qco.z %q4 : !qco.qubit -> !qco.qubit
%q6 = qco.h %q5 : !qco.qubit -> !qco.qubit

qco.sink %q6 : !qco.qubit
```

after-cancellation.mlir

```
%q0 = qco.alloc : !qco.qubit

%q1 = qco.h %q0 : !qco.qubit -> !qco.qubit
%q2 = qco.z %q1 : !qco.qubit -> !qco.qubit

%q3 = qco.z %q2 : !qco.qubit -> !qco.qubit
%q4 = qco.h %q3 : !qco.qubit -> !qco.qubit

qco.sink %q4 : !qco.qubit
```

FIGURE 6. Hybrid optimization through MLIR regions. A loop is implemented as an `scf.for` operation with a region containing `QCO` operations (left window). After loop unrolling exposes adjacent Hadamard gates across iteration boundaries (center window), the Hadamard-cancelation pattern removes them (right window) using the same rewrite logic as for flat circuits.

can instead reason about the first and last operations in the loop body and cancel Hadamards across consecutive iterations.

The same idea applies to back-end-aware compilation. `mqt-cc` includes a qubit mapping pass that assigns logical qubits to physical qubits on devices with limited connectivity and inserts the required SWAP gates.[11] Porting this algorithm to MLIR lifted it from flat circuits to programs with structured control flow, a capability that was architecturally absent from its original open-source implementation.[12]

Because hybrid programs use MLIR's standard infrastructure, they also inherit classical analyses such as constant folding, constant propagation, and dead-code elimination. These analyses can resolve loop bounds or branch conditions and thereby enable downstream quantum optimizations that need static information. At the same time, `QCO`'s value-semantics model keeps the quantum dataflow explicit: Transformations that preserve the def-use structure of qubit values preserve the corresponding circuit semantics. MLIR's pattern rewriting infrastructure guarantees that canonicalization patterns are applied atomically and consistently, so the compiler cannot produce a malformed hybrid program as a side effect of applying classical optimizations to the control flow. As quantum programs grow more complex—integrating with error decoders, classical optimization routines, and hardware control logic—`mqt-cc`'s correctness-by-construction property will become increasingly valuable.

CONCLUSION

Quantum computing needs software stacks that connect high-level applications to heterogeneous hardware without losing the program structure needed for compilation, optimization, and verification. Rather than rebuilding this infrastructure from scratch, the classical-first approach extends mature compiler frameworks with quantum-specific concepts and enables integration with existing scientific-computing environments. In this article, we have shown how this idea can be realized using the MQSS as a representative example, where frontend adapters connect established programming frameworks to a middle-end layer for scheduling and compilation, and QDMI provides a uniform interface to quantum back-end devices. At the compiler level, `mqt-cc` builds on MLIR through two complementary dialects: one designed for interoperability with external tools and another that explicitly represents quantum dataflow. Automated conversions connect both representations and allow the compiler to move between integration-friendly and optimization-friendly forms.

This design makes existing compiler infrastructure directly useful for quantum programs. Local gate optimizations become compact canonicalization patterns, while MLIR's region-based structure lets the same machinery apply inside functions, loops, branches, and nested hybrid programs. Classical passes can expose new quantum optimization opportunities, and back-end-aware passes can be lifted from flat circuits to structured programs. By adding quantum-specific semantics only where necessary and reusing classical infrastructure wherever possible, we demonstrated a path toward modular, interoperable, and correctness-aware quantum-classical compilation. The MQT Compiler Collection is publicly available as open source at https://github.com/munich-quantum-toolkit/core.

ACKNOWLEDGMENTS

The authors acknowledge funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program Grant Agreement 101001318 and Grant Agreement 101114305

("MILLENION-SGA1" EU Project), and the Munich Quantum Valley (MQV K5 + K7), which is supported by the Bavarian state government with funds from the Hightech Agenda Bayern Plus. This work was supported by the BMFTR under Grant 13N17298 and Grant 01MQ25001I, the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Grant 563402549 and Grant 563436708. Furthermore, this research was funded by BMIMI, BMWET, the State of Upper Austria and the State of Tyrol within the COMET module Quantum Algorithm Engineering (FFG Grant 923923) managed by the Austrian Research Promotion Agency FFG.

REFERENCES

1. C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, 2004, pp. 75–88, doi: 10.1109/CGO.2004.1281665.
2. S. Pakin, "Classical software for quantum computers," *Comput. Sci. Eng.*, vol. 28, no. 2, pp. 59–66, 2026, doi: 10.1109/MCSE.2026.3685564.
3. A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, "Open quantum assembly language," 2017, *arXiv:1707.03429*.
4. A. W. Cross *et al.*, "OpenQASM 3: A broader and deeper quantum assembly language," *ACM Trans. Quantum Comput.*, vol. 3, no. 3, pp. 1–50, 2022, doi: 10.1145/3505636.
5. A. Javadi-Abhari *et al.*, "Quantum computing with Qiskit," 2024, *arXiv:2405.08810*.
6. V. Bergholm *et al.*, "PennyLane: Automatic differentiation of hybrid quantum-classical computations," 2018, *arXiv:1811.04968*.
7. C. Lattner *et al.*, "MLIR: Scaling compiler infrastructure for domain specific computation," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, 2021, pp. 2–14, doi: 10.1109/CGO51591.2021.9370308.
8. L. Burgholzer, D. Haag, Y. Stade, D. Rovara, P. Hopf, and R. Wille, "The MQT compiler collection: A blueprint for a future-proof quantum-classical compilation framework," 2026, *arXiv:2604.08674*.
9. R. Wille *et al.*, "QDMI—Quantum device management interface: Hardware-software interface for the Munich quantum software stack," in *Proc. IEEE Int. Conf. Quantum Comput. Eng. (QCE)*, 2024, pp. 573–574, doi: 10.1109/QCE60285.2024.10411.
10. P. Hopf *et al.*, "Integrating quantum software tools with (in) MLIR," in *Proc. Supercomput. Asia Int. Conf. High Perf. Comput. Asia Pacific Region*, 2026, pp. 42–54.
11. M. Reumann, Y. Stade, R. Wille, and L. Burgholzer, "MLIR for quantum beyond gate cancellation: Quantum circuit mapping reimagined," 2026, *arXiv:2607.02616*.
12. A. Zulehner, A. Paler, and R. Wille, "An efficient methodology for mapping Quantum circuits to the IBM QX architectures," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 7, pp. 1226–1236, 2019, doi: 10.1109/TCAD.2018.2846658.

PATRICK HOPF is a doctoral candidate at the Technical University of Munich (TUM), 80333, Munich, Germany, and a software engineer at MQSC, 85748, Garching, Germany. His research interests include quantum circuit compilation and compiler design. Hopf received his M.Sc. degree in quantum science and technology at TUM. Contact him at patrick.hopf@tum.de.

YANNICK STADE is a doctoral candidate at the Technical University of Munich, 80333, Munich, Germany. His research interests include quantum computing, compiler design, and high-performance computing. Stade received an M.Sc. degree in computer science from the Julius-Maximilians-University Würzburg. Contact him at yannick.stade@tum.de.

DAMIAN ROVARA is a doctoral candidate at the Technical University of Munich, 80333, Munich, Germany. His research interests include domain-specific languages and quantum program representations, as well as development support for structured quantum programs through advanced testing and optimization methods. Rovara received his M.Sc. degree in computer science at The University of Edinburgh. Contact him at damian.rovara@tum.de.

DANIEL HAAG is a doctoral candidate at the Technical University of Munich (TUM), 80333, Munich, Germany, a software engineer at MQSC, 85748, Garching, Germany, and a member of the core development team for the Munich Quantum Toolkit (MQT). His research interests include the design of robust compilation frameworks and exchange formats (such as jeff) to enable seamless interoperability between different layers of the quantum software stack. Haag received his M.Sc. degree in physics at TUM. Contact him at daniel.haag@tum.de.

MATTHIAS REUMANN is a doctoral candidate at the Technical University of Munich (TUM), 80333, Munich, Germany, and a contributor to the Munich Quantum Toolkit (MQT). His

research interests include transpilation and compilation of hybrid quantum–classical programs in MLIR. Reumann received his M.Sc. degree in computer science at TUM. Contact him at matthias.reumann@tum.de.

LUKAS BURGHOLZER is a senior researcher at the Technical University of Munich, (TUM) 80333, Munich, Germany, and the CTO of MQSC, 85748, Garching, Germany. His research interests include opensource software, quantum compilation, and quantum software stacks. Burgholzer received his Ph.D. degree in computer science at Johannes Kepler Universität Linz. Contact him at lukas@mq.sc.

ROBERT WILLE is a full and distinguished professor at the Technical University of Munich, 80333, Munich, Germany, cofounder and CEO of MQSC, 85748, Garching, Germany, and scientific director at the Software Competence Center Hagenberg. His research interests include design automation, quantum compilation, and software stacks. Wille received his Ph.D. degree in computer science at University of Bremen. Contact him at robert.wille@tum.de.